3. Mid-Training & Post-Training

From CPT to Agentic RL: Foundations, recipes, and failure modes

Table of contents

0.1 Learning goals

Beyond memorizing definitions, interviews test whether you can **choose the right stage** (CPT vs SFT vs RL vs distill) and **anticipate failure modes**.

> Tip
>
> **ELI5:** *Training an LLM is like training a new hire: first you teach broad skills, then company-specific knowledge, then "how we talk to customers," and finally you coach them with feedback on real tasks.*

By the end of this chapter, you should be able to:

- Explain *why* mid-training (CPT) exists and how it differs from pretraining and SFT.
- Implement critical data strategies: **General Replay** for memory, **packing** for throughput, and **chat templates + masking** for instruction tuning.
- Apply **PEFT** (LoRA/QLoRA) strategies, including **multi-tenant serving** patterns.
- Compare alignment techniques: **PPO/RLHF** vs. **DPO/ORPO** vs. **Agentic RL** (e.g., **GRPO** and related variants).
- Design pipelines that optimize for **Reasoning (System 2)** using verifiers and process rewards.
- Build reliable **tool use** systems (schema correctness + tool selection + chaining).
- Apply **test-time scaling** (sampling, revision, verifier reranking) to boost reliability.
- Debug common regressions like **the stability gap**, **reward hacking**, and **template mismatch**.

0.2 The big picture: The life cycle of an LLM

This chapter organizes the landscape into a decision flow: *What's missing: knowledge, behavior, preferences, reasoning reliability, or cost?* Each corresponds to a different lever.

0.2.1 Interview framing: "Which knob would you turn?"

When you're given a product requirement, answer in this order:

1. **Define the target behavior** (format, safety, tool use, reasoning, latency).
2. **Diagnose the gap** (knowledge gap vs behavior gap vs optimization gap).
3. **Pick the cheapest effective lever** (prompt → SFT → DPO → RL → CPT → distill).
4. **Name the regression risks** (forgetting, reward hacking, template mismatch, latency).

> **Note**
>
> **ELI5:** *If the model doesn't know the facts, teach it with reading (CPT). If it knows facts but talks wrong, teach it with examples (SFT). If it needs to prefer "better" answers, use preferences/RL. If it's too slow/expensive, compress it.*

We treat training and inference as a pipeline of altering distributions *and* compute budgets.

```
flowchart LR
  A[Base Model  0] --> B{Domain gap?}

  B -->|Yes: Knowledge/Vocab/Context| C[Mid-training / CPT]
  B -->|No: Just behavior| D[SFT]

  C --> E[ _mid]
  E --> D
  D --> F[ _sft]

  F --> G{Alignment path}
  G -->|Chat/Style| H[DPO / ORPO / PPO]
  G -->|Reasoning/Math/Code| I[Agentic RL: GRPO / Self-training loops]

  H --> J[ _aligned]
  I --> J

  J --> K{Inference budget?}
```

```
K -->|Low| L[Direct decode]
K -->|High| M[Test-time scaling]

J --> N[Distillation / Quantization]
N --> O[ _deploy]
```

Note

Core mental model

- **Pretraining:** builds the engine (general capabilities).
- **Mid-training (CPT):** tunes the engine for terrain (domain knowledge, context-length priors, sometimes RL-compatibility).
- **Post-training (SFT/DPO/RL):** teaches the driver (behavior, safety, preference alignment).
- **Test-time scaling:** spends compute *at inference* to navigate complex routes (better reliability without retraining).

0.3 Data topology: the hidden interview topic

Most real failures come from feeding the right data through the **wrong loss topology**.

0.3.1 The three "topologies" you should be able to explain

- **Packing (CPT):** maximize tokens/GPU by concatenating documents to fill the context window.
- **Masking (SFT):** compute loss only where you want supervision (typically assistant tokens).
- **Rollouts (RL):** the policy generates full sequences; you score outcomes and optimize expected reward.

0.3.2 Quick mental check

Ask: *"Which tokens contribute gradients?"*
- CPT: **all tokens** (next-token loss)
- SFT: **assistant tokens only** (masked)
- DPO: **chosen vs rejected** (contrast)
- RL: **tokens sampled by the policy** (on-policy)

Tip

ELI5: *Same text, different learning: CPT learns to continue text, SFT learns to answer like a tutor, RL learns by trying things and getting a score.*

A common interview failure is not distinguishing how data is *constructed* and how loss is *applied*.

Tip

Packing vs masking vs rollouts

- **Packing:** concatenating docs to fill context (often used in pretraining/CPT; sometimes also used in SFT for throughput).
- **Masking:** loss only on assistant tokens (typical in SFT on chat transcripts).
- **Rollouts:** generating full sequences until EOS (RL/agentic RL). Enables exploration and verifier-based selection.

```
flowchart LR
  A[Raw text / docs] --> B[CPT: next-token loss (often packed)]
  C[Instruction chats] --> D[SFT: masked loss on assistant tokens]
  E[Preference pairs] --> F[DPO/ORPO: contrast chosen vs rejected]
  G[Prompts + Verifier/Env] --> H[Agentic RL: rollouts + scoring + optimize]
```

---

1 Phase 1: Mid-training / Continued Pretraining (CPT)

CPT is the workhorse for **domain specialization** and **long-context priors**. It's also the most common place to introduce regressions if you don't guardrail general capabilities.

Tip

ELI5: *CPT is "more reading": you keep training the model on domain documents so it picks up jargon and facts naturally.*

1.0.1 When CPT is the right tool

Use CPT when you see: - high perplexity on in-domain corpora, - systematic entity/jargon failures, - domain-specific formatting/structures (legal docs, codebases), - long-document understanding gaps.

1.0.2 CPT design checklist (interview-ready)

- **Data:** quality > quantity; de-dup, remove boilerplate, enforce doc boundaries.
- **Mixture:** start with replay (e.g., 80/20) and tune by regression gates.
- **LR:** typically much lower than initial pretraining peak LR.
- **Eval:** track both *domain gains* and *general regressions* continuously.

1.1 What it is (and isn't)

1.1.1 What CPT optimizes

CPT uses the **same next-token objective** as base pretraining, but on a **different distribution**.

- It tends to improve *knowledge recall* and *domain fluency*.
- It can also improve *tool-use* and *reasoning* indirectly when your domain data contains those patterns (e.g., code repos, math solutions).

1.1.2 What CPT is not

- Not instruction-following by itself: you can CPT a model into a great encyclopedia that still refuses to answer in JSON.
- Not a replacement for alignment: CPT can even make safety worse if your corpus contains unsafe patterns.

> **Note**
>
> **ELI5:** *CPT teaches what to say; SFT teaches how to say it to a user.*

Continued **next-token training** on a target distribution. It is necessary when the base model lacks:

- **domain knowledge** (facts/jargon/entity priors),
- **long-context priors** (document structure, long-range retrieval),
- and sometimes **RL-compatibility** for reasoning-style RL (model-family differences).

It is *not* SFT: SFT is about behavioral formatting and instruction following.

1.2 Optional: Tokenizer extension

Tokenizer extension can be worth it when your domain has many **high-frequency terms** that get split into many sub-tokens (e.g., rare drug names, API identifiers, legal citations).

Tip

ELI5: *Tokenizer extension is like adding new dictionary words so the model stops spelling domain terms letter-by-letter.*

1.2.1 When to extend

- **Decision rule:** measure **fragmentation rate** on a domain vocabulary list. If important terms routinely become 5+ tokens, you're wasting context and compute.
- **Signal:** you see frequent truncation/length issues, or the model mangles domain terms (broken identifiers, misspellings, bad citations).

1.2.2 How to extend (minimal-risk workflow)

1. Compile a vocabulary shortlist (top entities, terms, APIs).
2. Add tokens for the worst offenders.
3. Resize embeddings; initialize new token rows (random, or average of constituent sub-token embeddings).
4. Warm up: oversample examples containing the new tokens for the first phase of CPT.

1.2.3 Failure modes to mention

- **Undertrained tokens:** new tokens behave like noise early → mitigated via oversampling and warm-up.
- **Segmentation mismatch:** any downstream pipeline that tokenizes text must use the updated tokenizer.
- **Distribution shock:** adding tokens changes token counts and packing; re-check sequence length assumptions.

1.3 The stability gap

1.3.1 Why it happens

Early in CPT, gradients strongly adapt the model to the new distribution, often pushing it away from general-purpose representations.

Common contributors: - too-high learning rate, - low diversity domain data (narrow corpora), - insufficient general replay, - noisy/mislabeled documents (scrapes, templated boilerplate).

1.3.2 Debugging playbook

If general benchmarks drop sharply: - decrease LR and/or increase warm-up, - increase replay ratio, - improve data quality (dedup, filter spam), - add **regularization to the reference** (e.g., KL to base logits if available).

Note

ELI5: *Stability gap is like cramming for one exam and temporarily forgetting other subjects—replay is doing a little "general homework" to prevent that.*

CPT often causes an early dip in general performance before recovering.

- **Mitigation: General replay** (e.g., 80% domain / 20% replay as a starting point), plus regression gates.

```
# Pseudocode: CPT loop with packing + replay

### Practical add-ons (what interviews like)
- **Curriculum:** start with higher replay, then anneal toward more domain.
- **Domain mixing:** if multiple sub-domains, use adaptive sampling (upweight domains with h
- **Guardrails:** run a small regression suite every N steps; stop/rollback on big drops.

for step in range(T):
    batch = sample(D_domain) if rand() < p_domain else sample(D_replay)

    # Packing: concatenate docs to fill context (no padding; delimit with EOS)
    x = pack_sequences(batch, seq_len=L)

    # Next-token objective
    loss = cross_entropy(model(x[:, :-1]), x[:, 1:])
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()

    if step % eval_every == 0 and regression_failed():
        tune(p_domain="down", lr="down", data_quality="up")
```

Note

Mid-training to enable RL scaling (optional add-on)
Two useful framing papers for the "CPT → RL compatibility" story are:

- *OctoThinker: Mid-training Incentivizes Reinforcement Learning Scaling* (arXiv:2506.20512) [@octothinker_2025]

- *On the Interplay of Pre-Training, Mid-Training, and RL on Reasoning Language Models* (arXiv:2512.07783) [@interplay_pre_mid_rl_2025]

---

2 Phase 2: Supervised Fine-Tuning (SFT)

SFT turns a pretrained model into a usable assistant: it learns **roles, format, and tool schemas**.

> Tip
>
> **ELI5:** *SFT is "learning by example": you show the model lots of good conversations and it imitates them.*

2.1 The objective: formatting & behavior

2.1.1 What SFT changes most

SFT is extremely effective for: - instruction following (do X, don't do Y), - output formats (JSON, XML, markdown), - tool calling patterns (function arguments, schemas), - safety refusals and policy style.

2.1.2 What SFT changes least

SFT is weak for injecting broad factual knowledge (unless you have huge volumes, which starts to look like CPT).

2.1.3 Common failure modes

- **Over-refusal / under-refusal** from imbalanced safety data.
- **Length bias:** model learns to be overly verbose/terse depending on label distribution.
- **Template mismatch:** breaks role separation or tool call formatting.

> Note
>
> **ELI5:** *SFT is teaching "customer support etiquette," not teaching new encyclopedic facts.*

SFT teaches interaction style, tool schemas, and safety behavior. It is *not* the most efficient lever for injecting broad new knowledge.

2.2 The chat template trap

2.2.1 Why templates matter

Most modern checkpoints are trained with special control tokens (role separators, message boundaries). If your SFT data uses a different template, you create a train/test mismatch.

Practical guidance - Adopt the base model's official chat template (or a validated equivalent). - Be consistent across SFT, DPO, and RL rollouts. - For tool-use, include explicit tool result messages in the same template.

> **Tip**
>
> **ELI5:** *Chat templates are the "punctuation and grammar" the model expects—change them and the model gets confused.*

Models see control tokens, not literal `"User:"` / `"Assistant:"` strings.

- **Example (illustrative):** `<|im_start|>user\n{content}<|im_end|>\n`
- **Risk:** mismatched templates lead to broken behavior (poor role separation, weird completions, tool-call failures).

2.3 Implementation: user masking

2.3.1 Why we mask

If you compute loss on user tokens, the model learns to *predict the prompt* instead of focusing capacity on *answering*.

Practical variants - **Assistant-only masking:** most common for chat. - **Selective masking:** also supervise tool-call structure but not chain-of-thought (if you separate hidden reasoning). - **Span masking:** supervise only the JSON block for tool calling.

> **Note**
>
> **ELI5:** *Masking is grading only the student's answer, not grading the question they were asked.*

We mask user tokens to prevent the model from learning to *predict* the prompt.

```
# Pseudocode: SFT with masking (PyTorch-style)

### Interview extension: data collator
In practice you build a collator that:
- concatenates multi-turn messages with separators,
- generates `labels` that are `-100` on user/tool-result tokens,
- optionally enforces max length with truncation that preserves the assistant answer.

def compute_sft_loss(model, input_ids, labels):
    # labels: user tokens set to -100 (ignored by CrossEntropyLoss)
    logits = model(input_ids).logits
    shift_logits = logits[..., :-1, :].contiguous()
    shift_labels = labels[..., 1:].contiguous()
```

```
    return F.cross_entropy(shift_logits.view(-1, V), shift_labels.view(-1))
```

Tip

Interview one-liner
"In SFT, we mask the user prompt because we want the model to answer questions, not learn to ask them."

---

3 Phase 3: Parameter-Efficient Fine-Tuning (PEFT)

PEFT methods adapt a model without updating all weights, enabling faster iteration and multi-tenant serving.

Tip

ELI5: *PEFT is like adding a small "personality chip" on top of a big brain instead of retraining the whole brain.*

3.1 LoRA, QLoRA, and multi-tenancy

3.1.1 What LoRA does

[TODO]

3.1.2 Design knobs

- **Target modules:** attention projections, MLP projections, or both.
- **Rank (r):** higher rank → more capacity but more memory/latency.
- **Merge vs on-the-fly:** merge adapters for deployment or apply dynamically per request.

3.1.3 QLoRA

QLoRA keeps the base in 4-bit and trains adapters in higher precision, making large models feasible on limited GPUs.

3.1.4 Multi-tenancy patterns

- Serve **one base + many adapters**, route requests by tenant.
- Batch by adapter_id to avoid mixing overhead.

> **Note**
>
> **ELI5:** *LoRA learns small "adjustments" that steer the model without moving all its weights.*

- **LoRA:** low-rank adapters: (W' = W + A ×B).
- **QLoRA:** LoRA on a quantized (e.g., 4-bit) frozen base model to fit larger models on smaller GPUs.
- **Multi-tenant serving:** serve **1 base + N adapters** per customer/product (e.g., LoRAX-style patterns). This is a common SaaS system design topic.

> **Warning**
>
> **Common failure mode: adapter interference**
> Adapters can regress on shared prompts or bleed style across tenants if routing/versioning/constraints are not handled carefully (especially in batching).

---

4 Phase 4: Alignment (Chat & Style)

4.1 Reinforcement learning (RL) for LLMs (and how it relates to DPO)

Interviews often expect you to explicitly separate: - **Preference optimization (DPO/ORPO):** *offline* learning from labeled comparisons. - **RL (REINFORCE/PPO/GRPO-family):** *online / on-policy* learning from model rollouts scored by a reward model or verifier.

> **Note**
>
> **ELI5:** *DPO learns from "which answer is better?" examples; RL learns by "trying answers and getting a score."*

4.1.1 Unified objective (the one formula that explains most variants)

We have a **policy** ( ) *(the LLM), a* ***reward*** *(r(x, y)) (from a reward model, verifier, unit tests, etc.), and often a* ***reference policy*** *(* {ref}) to prevent drift. [TODO]

> **Tip**
>
> **ELI5:** *The KL term is a "leash" that stops the model from learning weird tricks to game the reward.*

```
flowchart LR
  X[Prompt x] --> P[Policy   generates y]
  P --> R[Reward model / verifier r(x,y)]
  R --> A[Compute advantage signal]
  A --> U[Update policy   (keep close to  ref)]
  U --> P
```

4.1.2 RL algorithm zoo (what you should be able to define in one minute)

Below are the common post-training RL variants and how they differ in practice.

Method	Core idea	Typical inputs	Key tradeoff	**ELI5 (one sentence)**
REINFORCE	Vanilla policy gradient using sampled returns; often with a baseline to reduce variance	prompts → sampled completions → scalar rewards	simplest but high-variance; needs many samples	*"Try answers, see the score, and nudge the model toward higher-scoring words."*
PPO	Clipped policy gradient + (often) value baseline; plus KL to a reference	on-policy rollouts + reward model/verifier	more stable updates; heavier infra (advantages/GAE, sometimes critic)	*"Make small safe updates so noisy rewards don't jerk the model around."*
GRPO	PPO-like but uses **group-relative** baselines from multiple samples per prompt; avoids an explicit critic	K samples per prompt + scorer	cheaper/more scalable; relies on good within-group ranking signal	*"Generate many attempts and learn from how each one compares to the group."*

Method	Core idea	Typical inputs	Key tradeoff	**ELI5 (one sentence)**
DR.GRPO ("GRPO Done Right")	Fixes GRPO biases caused by length/std normalizations; focuses on unbiased token efficiency	same as GRPO	improved stability/efficiency; still needs grouping + verifier	*"Same as GRPO, but it stops accidentally over/under-weighting certain questions or lengths."*
GSPO (Group Sequence Policy Optimization)	Uses **sequence-level** importance ratios and sequence-level clipping instead of token-level ratios	K samples per prompt + scorer	more stable, especially for long outputs and MoE; infra can be simpler	*"Treat the whole answer as one unit when deciding how much to update."*
DAPO	"Decoupled clip" + "dynamic sampling" to stabilize and improve efficiency at scale	rollouts + reward model + sampling buffer	better stability/diversity and training efficiency; more moving parts	*"Clip updates more intelligently and keep sampling the useful examples."*
DPO	Offline objective that pushes the model toward preferred completions without rollouts	preference pairs (chosen/rejected) + reference	simple/stable; no exploration beyond the dataset	*"From two answers, learn to prefer the one humans picked—no trial-and-error rollouts needed."*

Tip

Interview tip: For any method above, be ready to answer: *(1) what data it needs, (2) where the stability comes from, (3) where it breaks, (4) how*

you would debug it.

4.1.3 REINFORCE (the foundation)

REINFORCE is the "starting point" for many LLM RL methods: update the model to increase the log-probability of sampled tokens proportional to a reward signal.

- **Strength:** conceptually clean; minimal machinery.
- **Weakness:** high-variance gradients → slow unless you use lots of samples and good baselines.

> **Note**
>
> **ELI5:** *REINFORCE is like playing darts blindfolded: you can learn, but you need lots of throws unless you add good feedback/baselines.*

4.1.4 PPO (RLHF classic)

PPO adds stabilizers on top of REINFORCE: - **clipping** limits how much the policy can change per step, - **advantages** (often GAE) reduce variance, - **KL-to-reference** discourages reward hacking and mode collapse.

> **Tip**
>
> **ELI5:** *PPO is "step carefully toward better answers," not "jump to whatever got a high score once."*

4.1.5 GRPO (group-relative policy optimization)

GRPO-family methods typically: 1. sample (K) completions for a prompt, 2. score each completion, 3. compute a **relative baseline** within the group, 4. update the policy using those relative advantages (often with a KL anchor).

```
flowchart TB
  X[Prompt x] --> G[Sample K completions]
  G --> S[Score each completion]
  S --> B[Group baseline]
  B --> A[Relative advantages]
  A --> U[Update policy]
```

> **Note**
>
> **ELI5:** *GRPO is like grading on a curve: you learn from how each attempt ranks among your own attempts.*

4.1.6 DR.GRPO (bias fixes)

In practice, GRPO can inadvertently overweight certain prompts or lengths depending on how you normalize by token count and how you scale advantages by within-group variance. "DR.GRPO" is a commonly cited set of fixes that reduce those biases.

> **Tip**
>
> **ELI5:** *DR.GRPO is GRPO with the "math accounting" fixed so you don't accidentally learn from the wrong thing.*

4.1.7 GSPO (sequence-level policy optimization)

GSPO shifts key operations from the token level to the **sequence level**: - the importance ratio is based on sequence likelihood, - clipping and optimization are done per sequence.

This can improve stability (especially for long-form completions and MoE RL training).

> **Tip**
>
> **ELI5:** *GSPO updates based on whether the entire answer is more likely, instead of focusing on token-by-token ratios.*

4.1.8 DAPO (decoupled clip + dynamic sampling)

DAPO is a GRPO-family approach that emphasizes two levers: - **decoupled clipping** (often asymmetric clip bounds to preserve diversity / avoid collapse), - **dynamic sampling** (filtering/sampling strategies to prioritize informative rollouts).

> **Note**
>
> **ELI5:** *DAPO is "don't over-clip the good stuff, and keep training on the most useful attempts."*

4.1.9 DPO in the same frame (why it belongs here)

DPO is often taught alongside RL because it solves the same alignment problem under different constraints: - no rollouts, - no reward model training loop, - but also no exploration.

> **Note**
>
> **ELI5:** *DPO is RL without the "trying" step—just learn directly from which answer is preferred.*

4.1.10 Interview Q&A (rapid)

Q: When would you pick DPO over PPO/GRPO?
A: when you have strong preference pairs, want stable/simple training, and don't need exploration.

Q: When would you pick GRPO/GSPO/DAPO over DPO?
A: when correctness is verifiable and sampling multiple candidates can discover new high-quality trajectories beyond your dataset.

Q: What's the #1 RL failure mode?
A: reward hacking or a mis-specified verifier → mitigate with KL anchors, stricter rewards, and targeted evals.

4.2 Path A: DPO / ORPO (offline)

4.2.1 What DPO is optimizing

DPO trains a policy to prefer (y_w) over (y_l) without an explicit reward model, using a contrastive objective relative to a reference policy.

When DPO shines - you have good preference data coverage, - you want stability and simpler infra, - you don't need exploration beyond the dataset.

When DPO struggles - sparse tasks (math/code correctness) where "preference" "correct" - domains where the dataset is biased or low-diversity

> **Note**
>
> **ELI5:** *DPO is "pick the better of two answers and nudge the model toward it," without training a separate scorer.*

Optimizes policy directly on preference pairs ((y_w, y_l)).

- **Pros:** stable, memory efficient, easy to scale.
- **Cons:** no exploration; limited by dataset quality and coverage.

4.3 Path B: PPO / RLHF (online)

4.3.1 What PPO adds

PPO uses on-policy rollouts + a reward signal (often a reward model) to push the policy toward higher reward while limiting drift via KL.

Practical components - **Reward model** (RM): scores outputs. - **Reference policy**: defines the KL anchor. - **Value function / critic**: reduces variance (not used in GRPO-style).

Failure modes - reward hacking, - mode collapse, - excessive KL drift or over-regularization.

> ⚠ Warning
>
> **ELI5:** *PPO is "try an answer, get a score, and adjust," but with guardrails so the model doesn't become weird.*

Classic RM + PPO loop.

- **Pros:** can explore new solutions when the "right behavior" isn't in the dataset.
- **Cons:** complexity and instability; requires reward model training and KL control.

---

5 Phase 5: Tool use & RAG

Tool use is one of the most interview-relevant applications of post-training because it connects modeling to system design.

> 💡 Tip
>
> **ELI5:** *Tool use is teaching the model to stop guessing and instead call a calculator / database / API when needed.*

5.0.1 Core subproblems (name these in interviews)

1. **Tool selection:** which tool to call (or none)?
2. **Argument construction:** produce valid, schema-conformant inputs.
3. **Execution handling:** read tool outputs, recover from errors, and continue.
4. **Final response:** integrate evidence and cite sources.

5.0.2 Common engineering levers

- constrained decoding for JSON/schema,

- retries with repairs (self-heal loops),
- tool-use evaluation: success rate, schema validity, groundedness.

5.1 Tool use as a data problem

5.1.1 Trajectory format

A robust training example includes the *full loop*:

- (optional) plan / intent
- **tool call** (name + args)
- **tool result** (observation)
- **final answer** (grounded in observation)

5.1.2 Constrained decoding

For high-stakes tools, enforce validity at generation time (grammar / JSON schema), not just via training data.

> **Note**
>
> **ELI5:** *Constrained decoding is like putting the answer in a form with required fields so the model can't scribble nonsense.*

Tool use is typically learned via SFT on tool trajectories:

$$\text{Thought} \rightarrow \text{Call} \rightarrow \text{Result} \rightarrow \text{Answer}$$

Common failure modes and fixes:

- **Hallucinated tools/arguments:** model produces invalid JSON or wrong schema
  - **Fix:** constrained decoding; schema validators; training on *negative* examples (when *not* to call tools)
- **Tool overuse:** calls tools unnecessarily
  - **Fix:** counterexamples + explicit decision data (call vs don't call)
- **Chaining failures:** can call once, can't plan multi-step
  - **Fix:** multi-turn/trajectory data + agentic eval suites

> **Note**
>
> **RAG is tool use**
> RAG is simply tool use where the tool is a vector DB (retrieve docs), followed by grounded synthesis.

---

6 Phase 6: Reasoning & agentic RL (System 2)

Reasoning RL focuses on **task completion and correctness**, often with verifiers, unit tests, or deterministic checkers.

> Tip
>
> **ELI5:** *Reasoning RL is like giving the model practice problems and only rewarding it when the final answer checks out.*

This phase optimizes for **correctness** and **task completion** (math/code/tool agents), not just preference.

6.1 Reasoning as an evaluation target

6.1.1 What "reasoning" usually means in practice

In interviews, define reasoning operationally as a bundle of measurable behaviors:

- **decomposition** (subgoals),
- **verification** (checks),
- **self-correction** (revise when wrong),
- **planning** (sequence tool calls).

6.1.2 How to evaluate

- correctness rate on verifiable tasks,
- robustness under perturbations,
- calibration (knowing when unsure),
- tool-use success when reasoning requires tools.

> Note
>
> **ELI5:** *Reasoning is "show your work and catch your own mistakes," not just giving an answer fast.*

Cover reasoning as a *bundle* of behaviors:

- **Decomposition:** break task into sub-goals
- **Verification:** check intermediate steps or outcomes
- **Self-correction:** revise when a verifier flags an error
- **Planning:** decide tool calls and their order

6.2 Outcome vs process supervision

6.2.1 Tradeoffs you should be able to articulate

- **Outcome rewards (ORM):** cheap and robust, but sparse → can be slow to learn.
- **Process rewards (PRM):** dense learning signal, but expensive and can overfit to "style of reasoning."

6.2.2 Hybrid patterns

- outcome reward + best-of-N sampling + SFT on successful traces,
- PRM only for difficult subsets,
- verifier ensembles to reduce brittleness.

> **Tip**
>
> **ELI5:** *Outcome reward grades the final exam; process reward grades each step of the homework.*

- **Outcome (ORM):** did the test pass? did the answer match the key? (sparse signal; cheap; robust)
- **Process (PRM):** did step 1 make sense? (dense; expensive; can reduce reward hacking)

6.3 Self-training loops (STaR/ReST-style)

Self-training is the simplest "agentic RL" pattern when you have a **verifier**: generate multiple candidates, keep the ones that pass checks, and train on them.

> **Note**
>
> **ELI5:** *STaR/ReST is like letting the model try many times, keeping the correct attempts, and studying those.*

6.3.1 Why self-training works

If correctness is **verifiable** (unit tests, exact match, deterministic checks), then sampling gives you a pool of attempts where some are correct even if the average attempt is not. Training on the verified subset increases the probability mass on successful trajectories.

6.3.2 A practical pipeline

1. **Generate** (K) candidates per prompt (often with higher temperature for diversity).
2. **Verify** each candidate (tests/checkers/verifier model).

3. **Filter** to positives (and optionally hard negatives).
4. **Train** the policy:
   - SFT on positives (behavior cloning), and/or
   - DPO with positives as "chosen" and negatives as "rejected".

6.3.3 Key knobs (interview-friendly)

- **K (samples per prompt):** higher (K) increases chance of at least one correct, but costs more compute.
- **Verifier precision:** false positives poison training; prefer strict checks early.
- **Diversity controls:** temperature/top-p and prompt variations prevent overfitting.
- **Curriculum:** start with easier problems; gradually introduce harder ones.

6.3.4 Failure modes

- **Self-confirmation loops:** verifier is weak → model learns wrong patterns that still "look good."
- **Mode collapse:** too much filtering → dataset becomes narrow; keep diversity.
- **Distribution drift:** model changes → regenerate trajectories periodically ("on-policy" refresh).

7 Pseudocode: self-training loop (STaR/ReST-like)

solutions = model.generate(problems, num_return_sequences=N) rewards = verify_solutions(solutions) # tests / oracle / checker gold = [s for s, r in zip(solutions, rewards) if r == 1.0] loss = sft_loss(model, gold) # or DPO with gold vs failed loss.backward() optimizer.step() optimizer.zero_grad()

```
## GRPO (Group Relative Policy Optimization)

### The core idea
GRPO removes the critic/value model by normalizing rewards **within a group** of samples for

- Sample \(K\) outputs per prompt.
- Score them with a verifier/reward model.
- Compute advantages via within-group normalization (e.g., \(r_i -  ext{mean}(r)\)).
- Update the policy using those normalized advantages.

### When it's attractive
- large models where a critic is expensive,
- verifiable tasks where you can sample multiple candidates,
```

```
- memory-constrained RL setups.

::: callout-tip
**ELI5:** *GRPO is "compare answers to each other for the same question" instead of learning
:::


GRPO (popularly cited via DeepSeekMath) uses a group-relative baseline and avoids a learned

- **Key idea:** sample a *group* of outputs for the same prompt and normalize rewards within
- **Benefit:** large memory savings → enables RL at larger model sizes.

```mermaid
flowchart TB
    subgraph Sampling
      P[Prompt x] --> G[Generate group {y1..yK}]
    end

    subgraph Scoring
      G --> V[Verifier / Reward model]
      V --> S[Scores {r1..rK}]
    end

    subgraph Optimization
      S --> M[Baseline: mean(r)]
      S --> A[Advantage: r_i - mean(r)]
      A --> U[Update policy ]
    end

    Sampling --> Scoring --> Optimization
```

7.0.1 GRPO and newer variants (recommended coverage)

Suggested topics to cover (interview-friendly, not exhaustive):

- **Group size (K):** bigger (K) → better ranking signal but more sampling cost.
- **Baselines:** mean vs median vs rank-based advantages.
- **Clipping/regularization:** PPO-style clipping and KL-to-reference to avoid drift.
- **Reward shaping:** mixing sparse outcome checks with heuristic partial credit.
- **RLOO / group baselines:** leave-one-out baselines to reduce bias.
```

> **Note**
>
> **ELI5:** *Newer GRPO variants mostly change "how we compute the baseline" and "how we prevent the model from drifting too far."*

Add a short "variants" subsection so candidates can speak to recency:

- **Off-policy / replay-friendly GRPO:** extending group-relative methods to off-policy updates (clipped objectives to stabilize drift).
- **Multi-objective GRPO:** handling multiple rewards (correctness + safety + style) with normalization/weighting to reduce reward hacking.
- **Agentic GRPO:** grouping by *states* or tool-step segments to improve credit assignment over long horizons.

```
# Pseudocode sketch: GRPO-style within-group normalization
outs = sample_k(policy, prompt=x, K=K)
scores = verifier(outs)           # or reward model
baseline = scores.mean()
advantages = scores - baseline
policy_update(policy, outs, advantages)  # PPO-like surrogate without critic
```

> **Warning**
>
> **Reward hacking** RL will exploit reward model/verifier blind spots. Defense-in-depth: - stronger verifiers, - held-out adversarial eval, - constraints (format/safety rules), - process supervision (when feasible).

---

8 Phase 7: Test-time scaling

Test-time scaling spends compute **during inference** to increase reliability, without retraining the model.

> **Tip**
>
> **ELI5:** *Test-time scaling is like thinking twice: generate several attempts, check them, and pick the best.*

8.0.1 Common patterns to mention

- best-of-N + verifier,
- self-consistency voting,
- critique → revise loops,
- search (Tree-of-Thought, MCTS) with pruning.

Spending compute *at inference* to improve reliability.

1. **Best-of-N:** sample N candidates, score with a verifier, take the best.
2. **Sequential revision:** draft → critique → fix.
3. **Search:** Tree-of-Thought / MCTS-style exploration (most useful for agents and tool chains).

```
# Pseudocode: test-time scaling (best-of-N)
cands = model.generate(prompt, n=16, temp=0.7)
scores = verifier.score(prompt, cands)
best_response = cands[argmax(scores)]
```

> Tip
>
> **Interview framing**
> Test-time scaling is an "inference lever" when retraining is expensive or slow. The tradeoff is latency/cost.

---

9 Phase 8: Distillation

Distillation compresses behaviors from a strong teacher into a cheaper student, often preserving much of the capability at lower cost.

> Tip
>
> **ELI5:** *Distillation is teaching a smaller model by letting it copy a smarter model's homework.*

9.1 Black-box vs white-box

9.1.1 Black-box (most common in practice)

- call a teacher API to generate high-quality traces (solutions, tool trajectories, reasoning).
- train the student on those traces (SFT) or preference pairs (DPO).

9.1.2 White-box (when you have weights)

- match teacher logits (KL) for smoother learning signal.
- can combine with response distillation.

9.1.3 Practical tips

- filter low-quality teacher outputs; keep diversity.

- decide whether to include chain-of-thought or only summarized reasoning (policy/compliance dependent).
- use curriculum: easy → hard.

> **Note**
>
> **ELI5:** *Black-box distillation learns from the teacher's final answers; white-box distillation also learns from the teacher's "confidence" (logits).*

- **Black-box:** teacher generates CoT/tool traces → student learns from traces (SFT / DPO).
- **White-box:** student matches teacher logits (requires weight access).

---

10 Capstone: The "Recipes" cheat sheet

10.0.1 How to use this table in interviews

Start from the product constraint, pick the simplest effective lever, then name the failure mode and the mitigation. Interviewers reward structured thinking.

> **Tip**
>
> **ELI5:** *The cheat sheet is a "choose-your-own-adventure": pick the training stage that fixes your specific problem with the least risk.*

Problem	Recommended phase	The recipe	Key failure mode
Model lacks jargon	**CPT**	80% domain / 20% replay (start) + gates	catastrophic forgetting
Strict JSON schema	**SFT + tooling**	correct template + masking + schema tests	template mismatch
Many customers	**PEFT**	QLoRA + multi-tenant serving	adapter interference
Math/code logic	**Agentic RL / reasoning**	verifiers + GRPO/variants + anti-hacking eval	reward hacking
High reliability	**Inference**	test-time scaling + verifier reranking	latency/cost explosion

11 End-of-chapter drills

Add a habit: answer drills with (1) diagnosis, (2) proposed lever, (3) risks, (4) measurements.

> **Note**
>
> **ELI5:** *A good system answer is "what I'd change, why, what could break, and how I'd know."*

1. **Design:** Build a "Medical Scribe" that knows rare drug names (CPT) but refuses to prescribe (policy + eval gates).
2. **Systems:** Explain how **GRPO** saves memory compared to PPO and why that matters for training 70B+ models.
3. **Tradeoff:** You have a fixed compute budget. Do you spend it on **DPO** (training) or **test-time scaling** (inference)?
4. **Debug:** Your SFT model answers correctly but formats the tool call wrong. Do you add more data or switch to constrained decoding?
5. **Agentic:** Your agent fails on multi-step tool chaining. What do you change in data (trajectories), reward/verifier design, and test-time search?

11.1 Appendix: References (suggested BibTeX keys)

You can populate `references.bib` with entries for these keys:

- `lora_2021`, `qlora_2023`
- `dpo_paper_2023`
- `deepseek_math_2024` (GRPO)
- `revisiting_grpo_2025` (GRPO variants; off-policy extensions)
- `toolformer_2023`, `toollmm_2023`, `gorilla_2023`
- `verify_step_by_step_2024`, `math_shepherd_2023`
- `stability_gap_2024`, `stability_gap_acl_2025`
- `scaling_test_time_compute_agents_2025`
- `octothinker_2025`, `interplay_pre_mid_rl_2025`